1-5-2008

Putting the Automatic Back into AD: Part I, What's Wrong (CVS: 1.1)

Jeffrey M. Siskind
Purdue University, qobi@purdue.edu

Barak A. Pearlmutter
barak@cs.nuim.ie

Putting the Automatic Back into AD: Part I, What's Wrong (CVS: 1.1)

Jeffrey Mark Siskind[1] and Barak A. Pearlmutter[2]

[1] School of Electrical and Computer Engineering, Purdue University, 465 Northwestern Avenue, West Lafayette IN 47907-2035 USA qobi@purdue.edu
[2] Hamilton Institute, National University of Ireland Maynooth, Co. Kildare, Ireland barak@cs.nuim.ie

Summary. Current implementations of automatic differentiation are far from automatic. We survey the difficulties encountered when applying four existing AD systems, ADIFOR, TAPENADE, ADIC, and FADBAD++, to two simple tasks, minimax optimization and control of a simulated physical system, that involve taking derivatives of functions that themselves take derivatives of other functions. ADIC is not able to perform these tasks as it cannot transform its own generated code. Using FADBAD++, one cannot compute derivatives of different orders with unmodified code, as needed by these tasks. One must either manually duplicate code for the different derivative orders or write the code using templates to automate such code duplication. ADIFOR and TAPENADE are both able to perform these tasks only with significant intervention: modification of source code and manual editing of generated code. A companion paper presents a new AD system that handles both tasks without any manual intervention yet performs as well as or better than these existing systems.

Key words: Nesting, multiple transformation, forward mode, ADIFOR, TAPENADE, ADIC, FADBAD++

1 Introduction

The hallmark of Automatic Differentiation is that it is—or at least should be—*automatic*. One wishes to take derivatives of unmodified programs with minimal, and ideally no, manual intervention. In this paper, we demonstrate how far we are from this ideal. (See [9] for another viewpoint.) We present two simple mathematical tasks, collectively coded in under 300 lines, code both tasks in FORTRAN, C, and C++, and relate our experiences in getting them to run with ADIFOR [2], TAPENADE [4], ADIC[3], and FADBAD++[1]. We were able to run these programs under ADIFOR and TAPENADE only by modifying the source code in different fashions that are specific to each preprocessor and with significant manual editing of the output of TAPENADE. We discovered that using FADBAD++ one cannot compute derivatives of different orders with unmodified code, as needed by these tasks. One must either manually duplicate code for the different derivative orders or write the code using templates to automate such code duplication. Finally, we discovered that it is not possible to perform either of these tasks with ADIC at all.

The central limitation discovered in all of these systems in the inability to nest. Nesting is fundamental to programming: one expects to be able to nest conditionals inside conditionals,

do loops inside do loops, etc. In the context of AD, this corresponds to taking derivatives of functions that take derivatives. In Sect. 2 we give two realistic tasks where such nesting is crucial: minimax optimization to find a saddle point of a function, as is done in game theory, and determining an optimal value of a control parameter for a simulated physical system, as is done in automatic control. See [5] for another recent application of nesting. For transformation-based AD systems, like ADIFOR, TAPENADE, and ADIC, such nesting is accomplished by transforming the transformed code produced by the preprocessor, i.e., passing code through the preprocessor multiple times. For overloading-based AD systems, like FADBAD++, such nesting is accomplished by overloading the overloaded operators.

ADIFOR, TAPENADE, and ADIC all provide a mechanism to allow the user to change the naming convention of differentiated components of programs. It appear that this feature was included in these systems precisely to support nesting, i.e., transformation of transformed code. It is necessary to avoid conflating the tangents of different derivatives as could otherwise potentially occur when nesting derivatives [7, 8]. We know of no other use for this feature. While we make crucial use of this feature in the tasks in Sect. 2, this feature alone is not sufficient to support transformation of transformed code.

The authors the above systems are aware of the issues involved with nesting. A paper [2] on ADIFOR states on p. 18:

> While we currently can just process the ADIFOR-generated code [...]

The TAPENADE FAQ at `http://www-sop.inria.fr/tropics/` states:

> For example, one can use the forward mode twice, to get directional second derivatives. We know of some people who have tried that with TAPENADE, and apparently it worked.
> [...]
> However this requires a bit more interaction with the end-user.
> [...]
> The idea to obtain second derivatives is to apply Automatic Differentiation twice. Starting from a procedure `P` in file `p.f` that computes $y = f(x)$, a first run of TAPENADE e.g., in tangent mode through the command line:
>
> ```
> $> tapenade -d -head P -vars "x" -outvars "y" p.f
> ```
>
> returns in file `p_d.f` a procedure `P_D` that computes $yd = f'(x).xd$. Now a new run of TAPENADE on the resulting file e.g., in tangent mode again through the command line:
>
> ```
> $> tapenade -d -head P_D -vars "x" -outvars "yd" p_d.f
> ```
>
> returns in file `p_d_d.f` a procedure `P_D_D` that computes $ydd = f''(x).xd.xd0$. Specifically if you call `P_D_D` with inputs $xd = 1.0$ and $xd0 = 1.0$ in addition to the current x, you obtain in output ydd the second derivative $f''(x)$.
> [...]
> Doing this, you might encounter a couple of simple problems that you will need to fix by hand like we usually do:
>
> - The first multi-directional differentiation creates a program that includes a new file `DIFFSIZES.inc`, containing information about array sizes. Precisely, the include file must declare the integer constant `nbdirsmax` which is the maximum number of differentiation directions that you plan to compute in a single run. `nbdirsmax` is used in the declarations of the size of the differentiated arrays. You must create this file `DIFFSIZES.inc` before starting the second differentiation step. For instance, this file may contain

```
integer nbdirsmax
parameter (nbdirsmax = 50)
```

if 50 is the max number of differentiation directions. If what you want is the Hessian, this max number of differentiation directions is the cumulated sizes of all the inputs *x*.

- The second multi-directional differentiation requires a new maximum size value `nbdirsmax0`, which is *a priori* different from `nbdirsmax`. For the Hessian case, it is probably equal to `nbdirsmax`. What's more, TAPENADE has inlined the 1st level include file, so what you get is a strange looking piece of declarations:

```
      ...
      INCLUDE 'DIFFSIZES.inc'
C Hint: nbdirsmax should be the maximum ...
      INTEGER nbdirsmax
      PARAMETER (nbdirsmax=50)
      ...
```

  We suggest you just remove the `INCLUDE 'DIFFSIZES.inc'` line, and hand-replace each occurrence of `nbdirsmax0` by either `nbdirsmax` or even 50!

We reported one of the issues discussed in Sect. 5 relating to ADIC and received this Email from Paul Hovland in response:

> This is a known issue, but not one that we've thought very hard about how to work around because we haven't had a compelling application. I think we can come up with a workaround, but we'll need to discuss it for a while. One of us will try to get back to you later in the week.

The FADBAD++ web site at `http://www2.imm.dtu.dk/~km/FADBAD/` states:

> **Combinations of automatic differentiation**
> One of the very unique things of FADBAD++ is the ability to compute high order derivatives in a very flexible way by combining the methods of automatic differentiation. These combinations are produced by applying the templates on themselves. For example the type `B< F< double > >` can be used in optimisation for computing first order derivatives by using the backward method and second order derivatives by using a backward-forward method.

The remainder of this paper demonstrates the distance between the above desiderata and current practice. A companion paper presents a new language and a new compiler that can process both of the tasks presented without any manual intervention and which generates code that is as fast as or faster than the above mentioned systems.

2 Tasks

We use two tasks to illustrate the nesting issues that arise with current AD implementations. Variants of both tasks appear in other papers, coded in different languages for different AD systems. For this paper, we coded each task in FORTRAN, C, and C++, for use by ADIFOR, TAPENADE, ADIC, and FADBAD++. The variants in the different languages differ only in ways specific to the language and the AD implementation. They share the same algorithms, structure, order, naming conventions, etc. In the next four sections, we use these two tasks as a running example to illustrate the nesting issues that arise with current AD implementations. For each task and each AD system, we went through a number of variants as we attempted to get the task working. Figure 1 gives the essence of the first variant for FORTRAN with ADIFOR. Length restrictions preclude including all variants of all

tasks for all AD systems in this paper. However, all of these variants are available from `http://www.bcl.hamilton.ie/~qobi/tr-08-02/`. That web site contains a `run` script for each variant that replicates the issues involved along with a file `run.text` giving the output of that script for each variant. We recommend that the reader make use of the 'Try' entries in that web site to follow the discussion in the next four sections.

Figure 1(a) gives the essence of the common code shared between these tasks. It omits the subroutines `vplus` and `vminus` that perform vector addition and subtraction, the subroutine `ktimesv` that multiplies a vector by a scalar, the subroutines `magnitude_squared` and `magnitude` that compute the magnitude of a vector, and the subroutines `distance_squared` and `distance` that compute the L_2 norm of the difference of two vectors. The subroutine `multivariate_argmin` implements a multivariate optimizer using adaptive naïve gradient descent. This iterates $\mathbf{x}_{i+1} = \eta \nabla f\ \mathbf{x}_i$ until either $\|\nabla f\ \mathbf{x}\|$ or $\|\mathbf{x}_{i+1} - \mathbf{x}_i\|$ is small, increasing η when progress is made and decreasing η when no progress is made.

Figure 1(b) contains the essence of the first task, `saddle`, that computes the saddle point of a function:

$$\min_{(x_1,y_1)} \max_{(x_2,y_2)} ({x_1}^2 + {y_1}^2) - ({x_2}^2 + {y_2}^2)$$

(It omits the code for `gradient_outer` as this code is analogous to the code for `gradient_inner`.) This task is a variant of an example from [6], differing in that it uses forward AD instead of reverse AD to compute gradients and naïve gradient descent instead of gradient descent with a line search.

Figure 1(c) contains the essence of the second task, `particle`, a variant of an example from [8] where the textbook Newton's method for optimization has been replaced with naïve gradient descent. (It omits the code for `gradient_p` and `gradient_naive_euler` as this code is analogous to the code for `gradient_inner`.) This task models a charged particle traveling nonrelativistically in a plane with position $\mathbf{x}(t)$ and velocity $\dot{\mathbf{x}}(t)$. The particle is accelerated by an electric field formed by a pair of repulsive bodies,

$$p(\mathbf{x}; w) = \|\mathbf{x} - (10, 10 - w)\|^{-1} + \|\mathbf{x} - (10, 0)\|^{-1}$$

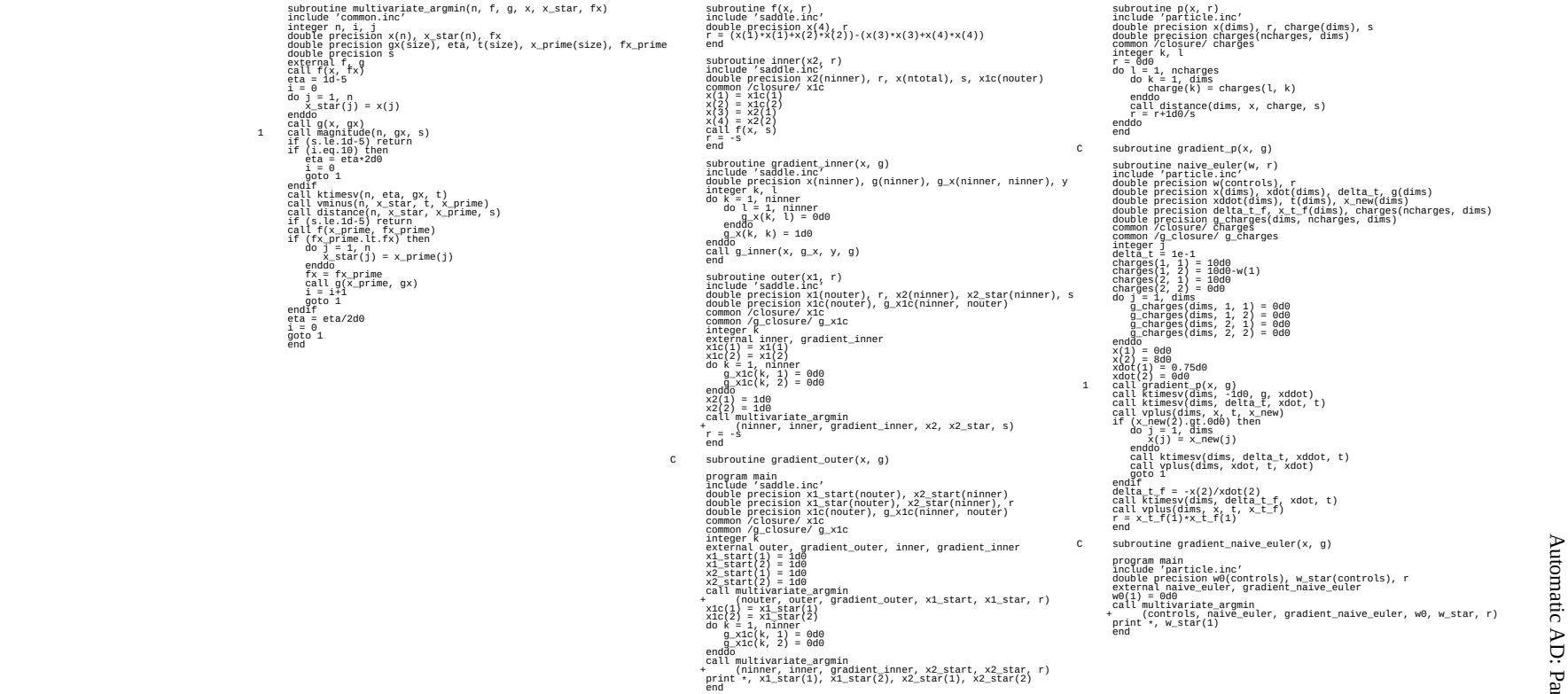
where w is a modifiable control parameter of the system, and hits the x-axis at position $\mathbf{x}(t_f)$. We optimize w so as to minimize $E(w) = x_0(t_f)^2$, with the goal of finding a value for w that causes the particle's path to intersect the origin. We use Naïve Euler ODE integration:

$$\begin{aligned} \ddot{\mathbf{x}}(t) &= -\nabla_{\mathbf{x}}\, p(\mathbf{x})|_{\mathbf{x}=\mathbf{x}(t)} \\ \dot{\mathbf{x}}(t + \Delta t) &= \dot{\mathbf{x}}(t) + \Delta t\, \ddot{\mathbf{x}}(t) \\ \mathbf{x}(t + \Delta t) &= \mathbf{x}(t) + \Delta t\, \dot{\mathbf{x}}(t) \end{aligned}$$

to compute the particle's path. We use linear interpolation to find the point where the particle hits the x-axis:

$$\begin{aligned} \text{When } x_1(t + \Delta t) &\leq 0 \\ \text{let: } \Delta t_f &= -x_1(t)/\dot{x}_1(t) \\ t_f &= t + \Delta t_f \\ \mathbf{x}(t_f) &= \mathbf{x}(t) + \Delta t_f \dot{\mathbf{x}}(t) \\ \text{Error: } E(w) &= x_0(t_f)^2 \end{aligned}$$

We minimize E with respect to w using `multivariate_argmin`.



```
      subroutine multivariate_argmin(n, f, g, x, x_star, fx)
      include 'common.inc'
      integer n, i, j
      double precision x(n), x_star(n), fx
      double precision gx(size), eta, t(size), x_prime(size), fx_prime
      double precision s
      external f, g
      call f(x, fx)
      eta = 1d-5
      i = 0
      do j = 1, n
         x_star(j) = x(j)
      enddo
      call g(x, gx)
1     call magnitude(n, gx, s)
      if (s.le.1d-5) return
      if (i.eq.10) then
         eta = eta*2d0
         i = 0
         goto 1
      endif
      call ktimesv(n, eta, gx, t)
      call vminus(n, x_star, t, x_prime)
      call distance(n, x_star, x_prime, s)
      if (s.le.1d-5) return
      call f(x_prime, fx_prime)
      if (fx_prime.lt.fx) then
         do j = 1, n
            x_star(j) = x_prime(j)
         enddo
         fx = fx_prime
         call g(x_prime, gx)
         i = i+1
         goto 1
      endif
      eta = eta/2d0
      i = 0
      goto 1
      end
```

(a)

```
      subroutine f(x, r)
      include 'saddle.inc'
      double precision x(4), r
      r = (x(1)*x(1)+x(2)*x(2))-(x(3)*x(3)+x(4)*x(4))
      end

      subroutine inner(x2, r)
      include 'saddle.inc'
      double precision x2(ninner), r, x(ntotal), s, x1c(nouter)
      common /closure/ x1c
      x(1) = x1c(1)
      x(2) = x1c(2)
      x(3) = x2(1)
      x(4) = x2(2)
      call f(x, s)
      r = -s
      end

      subroutine gradient_inner(x, g)
      include 'saddle.inc'
      double precision x(ninner), g(ninner), g_x(ninner, ninner), y
      integer k, l
      do k = 1, ninner
         do l = 1, ninner
            g_x(k, l) = 0d0
         enddo
         g_x(k, k) = 1d0
      enddo
      call g_inner(x, g_x, y, g)
      end

      subroutine outer(x1, r)
      include 'saddle.inc'
      double precision x1(nouter), r, x2(ninner), x2_star(ninner), s
      double precision x1c(nouter), g_x1c(ninner, nouter)
      common /closure/ x1c
      common /g_closure/ g_x1c
      integer k
      external inner, gradient_inner
      x1c(1) = x1(1)
      x1c(2) = x1(2)
      do k = 1, ninner
         g_x1c(k, 1) = 0d0
         g_x1c(k, 2) = 0d0
      enddo
      x2(1) = 1d0
      x2(2) = 1d0
      call multivariate_argmin
     +   (ninner, inner, gradient_inner, x2, x2_star, s)
      r = -s
      end

C     subroutine gradient_outer(x, g)

      program main
      include 'saddle.inc'
      double precision x1_start(nouter), x2_start(ninner)
      double precision x1_star(nouter), x2_star(ninner), r
      double precision x1c(nouter), g_x1c(ninner, nouter)
      common /closure/ x1c
      common /g_closure/ g_x1c
      integer k
      external outer, gradient_outer, inner, gradient_inner
      x1_start(1) = 1d0
      x1_start(2) = 1d0
      x2_start(1) = 1d0
      x2_start(2) = 1d0
      call multivariate_argmin
     +   (nouter, outer, gradient_outer, x1_start, x1_star, r)
      x1c(1) = x1_star(1)
      x1c(2) = x1_star(2)
      do k = 1, ninner
         g_x1c(k, 1) = 0d0
         g_x1c(k, 2) = 0d0
      enddo
      call multivariate_argmin
     +   (ninner, inner, gradient_inner, x2_start, x2_star, r)
      print *, x1_star(1), x1_star(2), x2_star(1), x2_star(2)
      end
```

(b)

```
      subroutine p(x, r)
      include 'particle.inc'
      double precision x(dims), r, charge(dims), s
      double precision charges(ncharges, dims)
      common /closure/ charges
      integer k, l
      r = 0d0
      do l = 1, ncharges
         do k = 1, dims
            charge(k) = charges(l, k)
         enddo
         call distance(dims, x, charge, s)
         r = r+1d0/s
      enddo
      end

C     subroutine gradient_p(x, g)

      subroutine naive_euler(w, r)
      include 'particle.inc'
      double precision w(controls), r
      double precision x(dims), xdot(dims), delta_t, g(dims)
      double precision xddot(dims), t(dims), x_new(dims)
      double precision delta_t_f, x_t_f(dims), charges(ncharges, dims)
      double precision g_charges(dims, ncharges, dims)
      common /closure/ charges
      common /g_closure/ g_charges
      integer j
      delta_t = 1e-1
      charges(1, 1) = 10d0
      charges(1, 2) = 10d0-w(1)
      charges(2, 1) = 10d0
      charges(2, 2) = 0d0
      do j = 1, dims
         g_charges(dims, 1, 1) = 0d0
         g_charges(dims, 1, 2) = 0d0
         g_charges(dims, 2, 1) = 0d0
         g_charges(dims, 2, 2) = 0d0
      enddo
      x(1) = 0d0
      x(2) = 8d0
      xdot(1) = 0.75d0
      xdot(2) = 0d0
1     call gradient_p(x, g)
      call ktimesv(dims, -1d0, g, xddot)
      call ktimesv(dims, delta_t, xdot, t)
      call vplus(dims, x, t, x_new)
      if (x_new(2).gt.0d0) then
         do j = 1, dims
            x(j) = x_new(j)
         enddo
         call ktimesv(dims, delta_t, xddot, t)
         call vplus(dims, xdot, t, xdot)
         goto 1
      endif
      delta_t_f = -x(2)/xdot(2)
      call ktimesv(dims, delta_t_f, xdot, t)
      call vplus(dims, x, t, x_t_f)
      r = x_t_f(1)*x_t_f(1)
      end

C     subroutine gradient_naive_euler(x, g)

      program main
      include 'particle.inc'
      double precision w0(controls), w_star(controls), r
      external naive_euler, gradient_naive_euler
      w0(1) = 0d0
      call multivariate_argmin
     +   (controls, naive_euler, gradient_naive_euler, w0, w_star, r)
      print *, w_star(1)
      end
```

(c)

Fig. 1. The essence of the baseline ADIFOR code for the `saddle` and `particle` tasks. (a) The common code shared between the tasks. (b) The code for the `saddle` task. (c) The code for the `particle` task.

3 ADIFOR

For `saddle`, we first try to perform the first transformation to generate the code for `g_inner` (Try 1). We get the error:

```
Procedure g_inner undefined: required by procedure gradient_inner in module saddle.f.

Procedure h_outer undefined: required by procedure gradient_outer in module saddle.f.
```

despite the fact that we specified `AD_TOP=inner` and none of `g_inner`, `h_outer`, `gradient_inner`, and `gradient_outer`, are reachable from `inner`. So we add `g_inner` and `h_outer` to `AD_EXCLUDE_PROCS` for the first transformation (Try 2). Now we get the error:

```
Recursive procedure set:
        multivariate_argmin.f.2
        outer
        multivariate_argmin
```

despite the fact that there really is no recursion, since the nested call to `multivariate_argmin` is to a transformed variant. So we add `multivariate_argmin` to `AD_EXCLUDE_PROCS` for the first transformation and the first transformation succeeds. We then try to perform the second transformation to generate the code for `h_outer`. This compiles successfully (Try 3), but gives the wrong answer:

```
1. 1. 8.24632483E-06 8.24632483E-06
```

Inspection of `h_saddle.f` reveals that ADIFOR generated incorrect derivative code for `h_outer`. We conjecture that it may be confused by the nested calls to `multivariate_argmin`. So we manually copy the code to make two versions of `multivariate_argmin` so that there is no potential for confusion. This compiles successfully (Try 4) and gives the correct answer, but does so only accidentally, as inspection of `h_saddle.f` reveals that ADIFOR has still generated incorrect derivative code for `h_outer`. We conjecture that it may be confused by the indirect subroutine call in the variant of `multivariate_argmin` that is differentiated. Se we manually specialize that variant to eliminate the indirect subroutine call. Again, this compiles successfully (Try 5) and gives the correct answer, but does so only accidentally, as inspection of `h_saddle.f` reveals that ADIFOR has still generated incorrect derivative code for `h_outer`. So we split `saddle.f` into three files: `saddle1.f`, which will be transformed in the first pass, `saddle2.f`, which will be transformed in the second pass, and `saddle.f`, which will not be transformed. Now we see that ADIFOR has generated correct derivative code that yields the correct answer (Try 6):

```
8.24632483E-06 8.24632483E-06 8.24632483E-06 8.24632483E-06
```

For `particle`, we first try to formulate the program as a single file, as this task requires no differentiation through nested or indirect subroutines calls. We rely on our experience with `saddle` and start by adding `g_p` and `h_naive_euler` to `AD_EXCLUDE_PROCS` for the first transformation and `h_naive_euler` to `AD_EXCLUDE_PROCS` for the second transformation. This compiles successfully (Try 1), but gives the wrong answer:

```
0.
```

Inspection of `h_particle.f` reveals that ADIFOR has generated incorrect derivative code for `h_naive_euler`. So we again rely on our experience with `saddle` and split `particle.f` into three files. Now we see that ADIFOR has generated correct derivative code that yields the correct answer (Try 2):

```
0.207191875
```

4 Tapenade

For `saddle`, we start with the same code as for the initial ADIFOR version, differing only in the naming and calling conventions for differentiated subroutines. First, we perform the first transformation to generate the code for `inner_gv`. Then, we create the file `DIFFSIZES.inc` as required by the output of the first transformation. Then, we perform the second transformation to generate the code for `outer_hv`. Then, we augment the file `DIFFSIZES.inc` as required by the output of the second transformation. However, despite issuing no errors or warnings, TAPENADE generates code that gives compiler errors (Try 1). Inspection of `saddle_hv.f` reveals that TAPENADE generated incorrect derivative code for `outer_hv`. We conjecture that, like ADIFOR, it may be confused by the indirect subroutine call in the variant of `multivariate_argmin` that is differentiated. Se we again manually specialize that variant to eliminate the indirect subroutine call. Again, the TAPENADE-generated code gives compiler errors. Thus, we create a `sed` script to fix these errors, as discussed in the above TAPENADE FAQ entry and find that TAPENADE has now generated correct derivative code that yields the correct answer (Try 2).

For `particle`, we again start with the same code as for the initial ADIFOR version, differing only in the naming and calling conventions for differentiated subroutines, perform the first transformation to generate the code for `p_gv`, create the file `DIFFSIZES.inc`, perform the second transformation to generate the code for `naive_euler_hv`, augment the file `DIFFSIZES.inc`, and create a `sed` script to fix the errors in the TAPENADE-generated code. This compiles successfully (Try 1), but gives the wrong answer:

```
0.
```

Inspection of `particle_hv.f` reveals that TAPENADE generated incorrect derivative code for `naive_euler_hv` because our code contains a subroutine call that modifies aliased arguments. While this violates the FORTRAN 77 standard, ADIFOR and G77 were nonetheless able to generate correct code for this task. Furthermore, while TAPENADE issued a warning, the TAPENADE FAQ only discusses how this affects reverse mode, not forward mode as used in this task. We modify our code to eliminate the aliasing violation, modify the `sed` script accordingly, and find that TAPENADE has now generated correct derivative code that yields the correct answer (Try 2).

5 ADIC

For `saddle`, we start with a straightforward translation of the FORTRAN code used for the initial ADIFOR version into C. Since we will need to transform `common.c` as part of the second transformation, we first try to transform this code. Compiling this code (Try 1) yields syntax errors. Inspection of `common.ad.c` indicates that ADIC has generated incorrect code for the transformation of `multivariate_argmin`:

```
void multivariate_argmin(int n,
                         void (*f)(double *, double *),
                         void (*g)(double *, double *),
                         double *x,
                         double *x_star,
                         double *fx) {…}

void g_multivariate_argmin (int n,void (*f)(DERIV_TYPE *,DERIV_TYPE *);
void (*g)(DERIV_TYPE *) {…}
```

We conjecture that, like ADIFOR and TAPENADE, ADIC cannot differentiate code with indirect function calls. Thus we again manually specialize `multivariate_argmin` to eliminate

the indirect function call. Furthermore, since ADIC does not use flow analysis to determine what code needs to be differentiated, and thus differentiates everything in a file, this necessitates splitting `saddle.c` into three files, as before. However, unlike before, the unspecialized version of `multivariate_argmin` must be moved from `common.c` to `saddle.c` since we will need to differentiate `common.c`.

We next try to transform `saddle1.c` twice, as is needed by this task. However, the output of the first transformation uses variables declared to be of type `DERIV_TYPE` and this is defined by the generated file `ad_deriv.h` to be:

```
typedef struct {
        double value;
        double grad[ad_GRAD_MAX];
} DERIV_TYPE;
```

Note that ADIC failed to prefix many of the above identifiers, despite the fact that we specified `prefix`, `var_prefix`, and `type_prefix`. It also failed to control the file-name prefixes. Thus we rename the files appropriately and create a `sed` script to edit the code generated by ADIC to manually prefix the unprefixed identifiers. Furthermore, since ADIC cannot process much of its own generated `ad_deriv.h`, we make a variant that contains just the bare essentials. The second transformation attempt, however, is unsuccessful (Try 2). Since `saddle1.c` contains only two extremely simple functions, we conjecture that ADIC is not able to transform transformed code and abandon our attempt at running our two tasks in ADIC.

We also point out a further difficulty we have encountered in transforming transformed code with ADIC. With ADIFOR and TAPENADE, the driver code is straight FORTRAN. (See the code for `gradient_inner` in Fig. 1(b).) With ADIC, the corresponding driver code must use primitives like `DERIV_val`, `ad_AD_SetIndepArray`, `ad_AD_SetIndepDone`, and `ad_AD_ExtractGrad`:

```
void gradient_inner(double *x, double *g) {
  g_DERIV_TYPE g_x[INNER], y;
  int k;
  for (k = 0; k<INNER; k++) DERIV_val(g_x[k]) = x[k];
  ad_AD_SetIndepArray(&g_x[0], INNER);
  ad_AD_SetIndepDone();
  g_inner(&g_x[0], &y);
  ad_AD_ExtractGrad(&g[0], y);}
```

Nesting requires transforming such drivers and we have not been successful in doing so.

This also affects global variables which are written and then read across differentiation boundaries. In our tasks, the function `outer` must initialize the tangents of `x1c` to zero and the function `naive_euler` must initialize the tangents of `charges` to zero. This is done using the primitive `ad_AD_ClearGrad`. However, `outer` and `naive_euler` must be transformed and a similar problem arises.

6 FADBAD

For FADBAD++, we describe `saddle` and `particle` jointly. Our initial FADBAD++ variants are similar to the initial ADIC variants, differing primarily in that the functions `magnitude_squared`, `magnitude`, `distance_squared`, `distance`, `multivariate_argmin`, `f`, `inner`, `outer`, `p`, and `naive_euler`, return results rather than modify arguments passed by reference, use of C++ I/O, and formulating the implementation of the drivers `gradient_inner`, `gradient_outer`, `gradient_p`, and `gradient_naive_euler` to use the FADBAD++ API instead of the ADIC API (Try 1). These variants, however, cannot be run, since they don't implement `g_inner`, `h_outer`, `g_p`, and `h_naive_euler`.

One way of providing these is to manually simulate the behavior of a preprocessor like ADIFOR, TAPENADE, or ADIC. We do this by making a copy of all of the functions in `common.cpp`, prefixing all function identifiers in this copy with `g_` and changing all instances of `double` in this copy to `F<double>`. We also manually apply such a process to selected portions of `saddle.cpp` and `particle.cpp`, making prefixed, type-lifted copies of certain functions and global variables, namely `f`, `x1c`, `inner`, `charges`, and `p`. This simulates the first transformation. To simulate the second transformation, we make copies of all of the functions in `common.cpp`, including the ones created by the first transformation, prefixing all function identifiers in this copy with `h_` and changing all instances of `double` in this copy to `F<double>`. This creates some identifiers prefixed with `h_g_` and some instances of the type `F<F<double>>` (which must be manually changed to the type `F<F<double> >`). We also manually apply such a process to selected portions of `saddle.cpp` and `particle.cpp`, including the portions created by the first transformation, namely `f`, `x1c`, `inner`, `g_f`, `g_x1c`, `g_inner`, `gradient_inner`, outer, `charges`, `p`, `g_charges`, `g_p`, `gradient_p`, and `naive_euler`. This compiles successfully, and gives the correct answer (Try 2). However, this is both inelegant and labor intensive. Thus we rewrite the code from Try 1 using templates. This compiles successfully, and gives the correct answer (Try 3). Note that this also requires modification of our original code.

7 Conclusion

The goal of AD is to be able to *automatically* take derivatives of *unmodified* programs. We are far from this goal, at least when considering nesting, i.e., taking derivatives of functions that take derivatives of other functions. All the systems that we have tried require manual modification of either the source code, the automatically generated code, or both. ADIFOR requires manual partitioning of the code into different files to be transformed different numbers of times. TAPENADE requires manual specialization of subroutines to eliminate indirect subroutine calls and manual post-editing of code that has been transformed multiple times. FADBAD++ requires either manual simulation of a transformation process or writing code using templates. And ADIC is not able to handle such nesting at all. Furthermore, along the path to solving these tasks, we encountered situations with both tasks using both ADIFOR and TAPENADE where incorrect derivative code was produced without warning or error, sometimes leading to subsequent compiler errors and sometimes, but not always, leading to incorrect computational results.

In this paper, we have rationally reconstructed the minimal path from our original intent to the solution of each task using each AD system. The actual process of producing these variants was very labor intensive and involved the exploration of many blind alleys that have been omitted. For example, before we specified `AD_EXCEPTION_FLAVOR = performance`, we needed to add `ehsfid` and `ehufDO` to `AD_EXCLUDE_PROCS`. Even then, transforming transformed code with ADIFOR yielded incorrect code that gave compiler errors due to redundant declaration of `g_ehfid`. And with long file names, ADIFOR generates calls to `ehsfid` with long Hollerith constants that extend past column 72, again giving compiler errors. We had to create `sed` scripts to post-edit the ADIFOR-generated code to remove these flaws. Similarly, TAPENADE generates pedantic warnings about code that compares floating point values for equality. But it itself generates code that triggers such warnings upon subsequent transformation of transformed code.

Even ignoring the flaws encountered, the documented mode of use for these systems is far from automatic, requiring manual specification, through script files, include files, and command line parameters, of things like the files to scan, the functions and variables to include or

exclude from the transformation process, the prefixes to use, and the dimensions of generated arrays. Furthermore, much of this information is specific to a particular AD system. Both the official mode of use, as well as the specific source-code changes and post-editing steps we had to employ to achieve success, vary significantly between ADIFOR and TAPENADE despite the fact that they both apply to FORTRAN77. The same occurred between ADIC and FADBAD++, despite the fact that our initial code for both of these systems was written in vanilla C(except for use of C++ I/O). (FADBAD++ can handle functions that return real values while ADIC requires returning results by mutating values passed by reference, much like ADIFOR and TAPENADE can only transform subroutines, not functions.) This is all further complicated by the fact that the different AD systems needed different code (in `outer`, `naive_euler`, and the `main` for `saddle`) to handle the initialization of the tangents of the variables `x1c` and `charges` that were implemented as common variables in FORTRAN and global variables in C/C++.

Our companion paper describes a novel language and a novel compiler that addresses the shortcomings described in this paper. We hope that this paper clarifies why we believe that the work described in the companion paper is novel and significant and addresses issues that are not addressed by current AD implementations.

Acknowledgement. This work was supported, in part, by NSF grant CCF-0438806, Science Foundation Ireland grant 00/PI.1/C067, and a grant from the Higher Education Authority of Ireland. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies.

References

1. Bendtsen, C., Stauning, O.: FADBAD, a flexible C++ package for automatic differentiation. Technical Report IMM–REP–1996–17, Department of Mathematical Modelling, Technical University of Denmark, Lyngby, Denmark (1996)
2. Bischof, C.H., Carle, A., Corliss, G., Griewank, A., Hovland, P.: Generating derivative codes from Fortran programs. Preprint MCS–P263–0991, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill. (1992). Also appeared as Technical Report 91185, Center for Research in Parallel Computation, Rice University, Houston, Tex.
3. Bischof, C.H., Roh, L., Mauer, A.: ADIC — An extensible automatic differentiation tool for ANSI-C. Software–Practice and Experience **27**(12), 1427–56 (1997). DOI 10.1002/(SICI)1097-024X(199712)27:12⟨1427::AID-SPE138⟩3.0.CO;2-Q. URL `http://www-fp.mcs.anl.gov/division/software`
4. Hascoët, L., Pascual, V.: TAPENADE 2.1 user's guide. Rapport technique 300, INRIA, Sophia Antipolis (2004). URL `http://www.inria.fr/rrrt/rt-0300.html`
5. Martinelli, M.: Second derivatives via tangent-on-tangent and tangent-on-reverse (2007). Programme of the Sixth Euro AD Workshop
6. Pearlmutter, B.A., Siskind, J.M.: Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. ACM Trans. on Programming Languages and Systems (2008). To appear
7. Siskind, J.M., Pearlmutter, B.A.: Perturbation confusion and referential transparency: Correct functional implementation of forward-mode AD. In: A. Butterfield (ed.) Implementation and Application of Functional Languages—17th International Workshop, IFL'05, pp. 1–9. Dublin, Ireland (2005). Trinity College Dublin Computer Science Department Technical Report TCD-CS-2005-60
8. Siskind, J.M., Pearlmutter, B.A.: Nesting forward-mode AD in a functional framework. Higher-Order and Symbolic Computation (2008). To appear

9. Willkomm, J.: AD tools from a user's perspective (2007). Programme of the Sixth Euro AD Workshop